12 | 项目结构：为什么要按领域组织文件夹结构？

2021-06-22 王沛

《React Hooks 核心原理与实战》 课程介绍 >

讲述：王沛
时长 17:45 大小 16.26M

你好，我是王沛。今天我们来聊聊如何组织项目文件夹结构。

很多同学在做项目开发时都会有这样的体验：项目初期，做得爽，进展快。但等功能做得差不多了，比如已经完成 90%，会突然感觉进展慢下来了。在剩下 10% 的功能开发上花的时间，可能比前面做 90% 的功能还要多。

这种现象，有人称之为“90-90 定律”。这就导致我们很难预估项目究竟什么时候才能做完。

产生这个现象的原因是多种多样的，但其中一个很重要的原因就在于，随着功能的增加，项目会变得越来越复杂。而在项目的后期，每增加一个新功能，或者修改已有的一些功

能，都会带来很大的工作量。毕竟在做这些新的改动时，你需要考虑它们会对已有的功能产生什么影响。

而要改善或者解决这个问题，关键就在于：**每增加一个新的功能，整个应用程序的复杂度不应该明显上升。**这样才能保证我们的应用程序始终可扩展，可维护。

那么怎么在 React 应用中做到这一点呢？这正是我们这节课要讨论的问题。

软件复杂度的根源：复杂的依赖关系

我们经常会说，某个项目看上去好复杂。那么这个“复杂”，到底该怎么定义呢？

如果仔细思考就会发现，当某个功能需要层层嵌套的模块依赖，那么即使开发时觉得思路很顺，但是自己再回头去看，或者要让别人理解某个功能实现，就不得不去翻阅很深的调用链。这就是让你觉得复杂的直接原因。

那么我们可以这么说，**软件复杂度的根源完全来自复杂的依赖关系**。

找到了问题的根源，接下来就是解决问题了。在功能不断增加的时候，我们该怎么避免线性地产生复杂度呢？

在寻找答案之前，我们不妨先想象两个场景。

第一个场景，我们需要开发一个前端的问卷调查页面。页面有三个选项，一个提交按钮，这些就是所有的 UI 元素了。

很显然，这是一个极其简单的场景。可能在你的团队中，交给一个新手，甚至让实习生去练手，无论他们采用什么技术框架，你都会十分放心。因为复杂度完全可控，那么即使代码写得再初级也没有关系，只要功能满足要求就可以了。

第二个场景，我们需要开发一个有 50 个页面的应用，功能很多。但是呢，每个页面都是独立的，所有功能都是在各自的页面内实现，彼此之间没有任何共享的模块。

可以看到，在这个场景中，即使增加了新的页面，但事实上，也没有增加太多复杂度。而且，在修改已有功能的时候，我们也不用担心会破坏其他页面。

现在，我们不妨把这两个场景放在一起进行对比。你会发现，场景二实际上就是由多个场景一堆砌而成的。每一个功能都足够独立，这样每个功能就很容易实现，而且也容易维护。照此来看，要隔离复杂度，那我们要做的就是把一个复杂的系统模块化，并把每个模块之间的依赖降到最低。这样，每当增加新的功能时，也就不会增加整个系统的复杂度了。

事实上，在我近 5 年的工作中，很大一部分时间都是在思考和解决这个问题：**如何降低依赖，让整个大型应用的复杂度始终在可控范围内？**

只有这样，在团队内，无论是代码写得比较初级的新手，还是总想尝试新技术新方式的探索者，再或者是代码写得很漂亮的老手。他们都能在各自的功能模块内完成业务功能，而且尽可能少地互相影响，从而始终保证整个架构的可扩展。

你可能会说，这个目标听上去有点理想化啊，毕竟模块之间不可能没有互相依赖。我完全理解这样的想法，但是我更想强调的是：只有先制定一个清晰且正确的目标，那么我们之后在实际项目开发中才能尽量动用各种手段去靠近这个目标。

那么既然要隔离复杂度，首先要做的就是在物理层面，让源代码能够按功能模块组织在一起，也就是要正确地按领域去组织你的项目文件夹结构。

按领域组织文件夹结构

如果你已经用 create-react-app 创建过一个项目，你会发现，默认的文件夹组织类似如下结构：

复制代码

```
import CommentList from './CommentList';
function ArticleView() {
  return (
    <div className="article-view">
      <MainContent />
      <CommentList />
    </div>
  );
}
```

也就是说，源代码是从技术角度进行划分的。这对于一个小的功能，比如只有一个页面的问卷调查，没有什么问题。但是对于一个大型的前端应用来说，这种做法就会导致整个应用的不可扩展。

我看到很多同学的做法是，在 components、actions、Hooks 等文件夹下，再按照功能进行分类。而这个分类的做法呢，经常是按照技术功能进行进一步划分，比如 table、modals、pages 等。这种做法其实会增加项目结构的复杂度，开发起来也很不方便，主要体现在两个方面。

一方面，对于一个功能，我们无法直观地知道它相关的代码散落在哪些文件夹中。

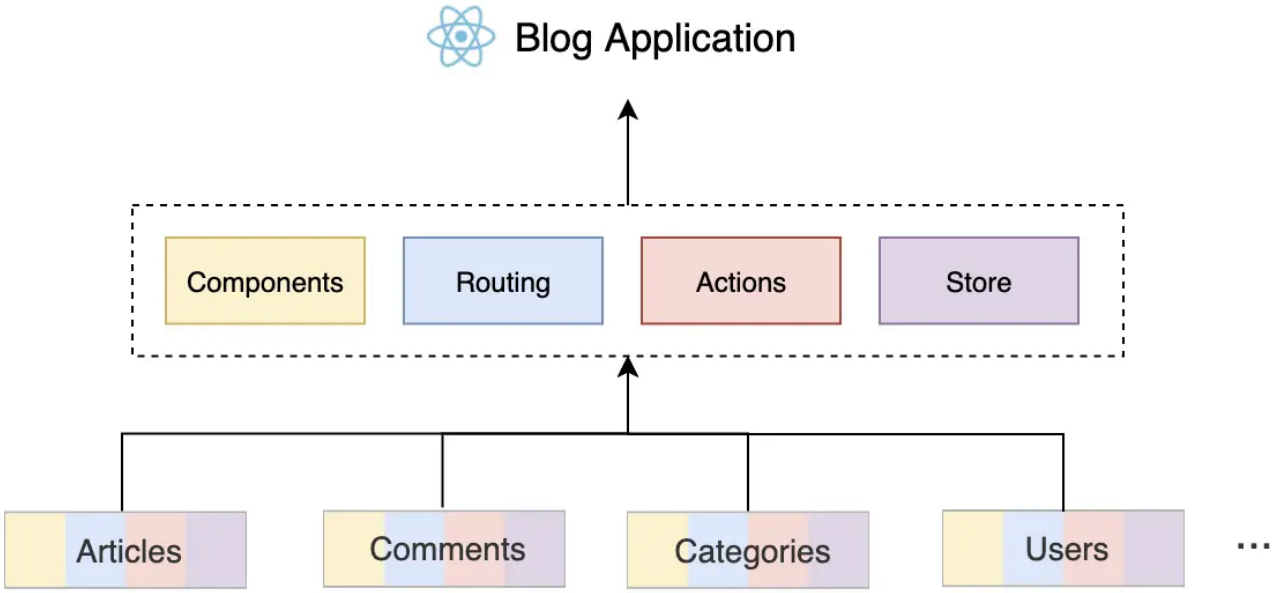
比如内容管理系统中的分类功能，可能有列表、下拉框、对话框、异步请求逻辑等，它们都在不同的文件夹中。

另一方面，开发一个功能时，切换源代码会非常不方便。

比如你在写分类列表功能时，就需要在组件、样式文件、action、reducer 等文件之间频繁地来回切换。而且，如果项目很大，那么你就需要展开很长的树结构，才能找到相应的文件，或者借助文件搜索去导航。不过，文件搜索导航的前提是，你还需要对整个功能的逻辑非常了解，知道有哪些文件。

产生这种开发难度的本质就在于，**源代码没有按照业务功能组织在一起，而是从技术角度进行了拆分**。所以呢，对于文件夹的组织，我们一定要**按领域去组织源代码**。一个与领域相关的文件夹，就类似于刚才讲的第一个场景，自身包含了自己需要的所有技术模块，这样无论是理解代码实现，还是开发时切换导航，都会非常方便。

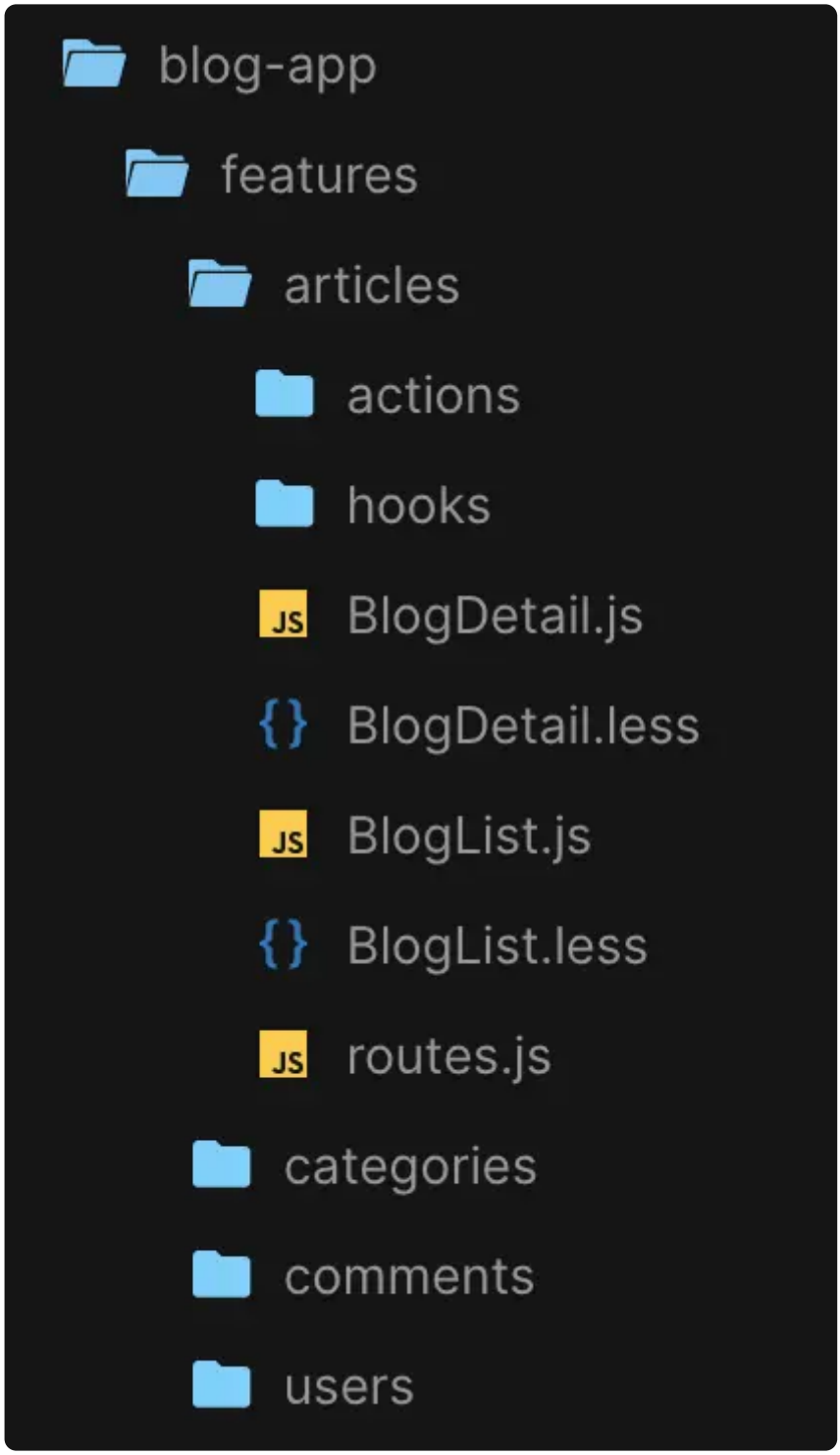
下面这张图描述了这样的项目结构，展示了对于一个博客应用，我们该如何组织项目代码：



可以看到，整个应用至少包含了文章管理、评论、分类、用户等模块。首先我们知道，一个 React 应用，一定是由一些技术部件组成的，比如 components、routing、actions、store 等，图中我用不同的颜色对这些部件进行了区分。

但是呢，如果我们将这些技术部件分散到不同的领域文件夹中，而每个领域文件夹都有自己的 compoents、routing、actions、store 等。这样的话，每一个文件夹就相当于一个小型的项目，包含了与自己相关的所有源代码，就便于理解和开发。

如下所示，演示了对于这样一个博客应用，文件夹的结构应该是什么样的：



可以看到，我们把业务模块的文件夹都放在了一个名为 features 的文件夹下，这样就可以和其它一些全局配置的代码区分开来。

一旦我们按照领域组织了项目的功能文件夹，那么，在每一个独立的功能下面，无论怎么组织源代码，都不会有太大的问题，因为都是很小的文件夹。比如在这张图中，我们把所有的组件和样式代码都放在了 articles 这样的文件夹下，而把 actions、hooks 等放到了独立的文件夹下。

同时呢，我们也要尽量扁平化地组织所有代码，而不是再去按小的功能去增加嵌套的文件夹。否则，如果你再回头去看代码，或者新加入的成员去看，会增加理解成本。

处理模块间的依赖

当我们通过文件夹对业务模块进行了隔离之后，接下来就要考虑**该怎样在模块之间进行交互**了。

这里需要注意的是，尽管各个模块的代码已经处在独立的文件夹之中了，但其实还是可以互相引用的，也就是可以任意依赖。

直观上来说，依赖是在代码中 import 了另外一个模块，但是如果对应到业务，那么它的本质是什么呢？仔细思考下，我们其实可以**把依赖从技术层面提升到业务层面**，也就是一个业务功能对另外一个业务功能的依赖。

从业务功能去理解，依赖可以分为两种。

第一种是硬依赖。如果功能 A 的实现必须基于功能 B，也就是说没有功能 B，功能 A 就是不可运行的，那么我们可以说 A 硬依赖于 B。

比如对于博客系统，评论功能肯定是基于文章功能的，因为评论的对象就是文章，脱离了文章，评论功能就没有意义。

第二种是软依赖。如果功能 B 扩展了功能 A，也就是说，没有功能 B，功能 A 自身也是可以独立工作的，只是缺少了某些能力。

同样对于博客应用，文章管理是主要的功能，而评论功能则可以认为是增强了文章的功能。照此来看，即使没有评论功能，文章功能也是可以独立运行的。这样就可以认为文章功能软依赖于评论功能。

在从业务功能上区分了两种依赖之后，我们还要落地到技术层面，也就是从技术角度对两者进行区分：**我们应该让文章功能相关的代码，不要硬依赖于评论功能的代码。**

什么是技术层面的硬依赖呢？比如说，在增加了评论功能之后，我们的文章页面可能就会使用如下的代码：

复制代码

```
import CommentList from './CommentList';
function ArticleView() {
  return (
    <div className="article-view">
      <MainContent />
      <CommentList />
    </div>
  );
}
```

可以看到，文章页面依赖了评论列表这个组件。如果评论列表组件不存在，那么文章页面也会无法显示。也就是说，当我们为应用增加了评论功能之后，就增加了技术组件的硬依赖，这样的话，系统的复杂度必然也会显著增加。

想象一下，按照这种做法，如果还要增加点赞功能、相册功能等，势必会导致复杂度不断上升，因为这些功能都需要在 ArticleView 这个组件内展示。

此外，如果你想去重构，或者移除某个功能，从技术角度来看，也是极为复杂的。比如说要移除评论功能，你就需要移除分散在各个领域的文件夹下的相关代码，就像这里的 ArticleView。

而我们要达到的目标其实是：**删除一个功能，就像删除一个文件夹那么简单。**这才是真正松耦合的系统。

所以呢，虽然在业务功能上是一个软依赖，但是在代码实现层面，却往往做成了硬依赖。这就导致随着功能的不断增加，整个应用变得越来越复杂，最终降低了整体的开发效率。

既然如此，我们就必须想办法，**让模块之间的交互不再通过硬依赖。**

在这里，我跟你介绍一种架构，这是我参与的项目基本上都会采用的，那就是**扩展点机制：在任何可能产生单点复杂度的模块中，通过扩展点的方式，允许其它模块为其增加功能。**

还拿博客文章这个例子来说，ArticleView 这个组件就是会产生单点复杂度的组件，因为它允许其它业务模块在文章页面上根据上下文数据去渲染额外的组件，那么我们可以使用扩

展点机制来隔离这种复杂度。示意的实现代码如下：

复制代码

```
function ArticleView({ id }) {
  const { article } = useArticle(id);
  return (
    <div className="article-view">
      <MainContent article={article} />
      {/* 定义了一个名为 article.footer 的扩展点 */}
      <Extension name="article.footer" args={article} />
    </div>
  );
}
```

在代码中，我们定义了一个名为 article.footer 的扩展点，允许其它模块为其贡献额外的 React 组件，使其在页面的底部渲染。接着，这个扩展点会将当前的 article 作为参数传递给额外的组件。这样，我们就可以在评论功能自己的文件夹内去扩展这个扩展点，从而具体渲染评论列表的组件了。

同样的，如果要增加点赞、打分等功能，我们也可以利用这个扩展点去实现 UI 的渲染。使用这个扩展点的评论列表模块，可以用类似的代码来实现：

复制代码

```
extensionEngine.register('article.footer', article => {
  return <CommentList article={article} />
});
```

可以看到，我们往扩展点引擎中注册了一个对 article.footer 这个扩展点的实现代码，从而可以在 渲染时被调用，最终渲染出评论列表。

同时呢，因为我们把这块逻辑可以隔离出来了，那就可以放在评论功能自己的文件夹下。如果文件夹被删除，那么文章页面也不会受影响，因为如果这个扩展点没有被任何逻辑扩展，那就不会渲染任何内容。

不过这里还有一个问题，我们该怎么去实现这样一个扩展点引擎呢？可能直觉上挺复杂，但其实并不困难，我们可以利用类似事件的订阅和发布模型去建立这样一个机制。

我自己呢，也将这个功能发布成了一个开源的 npm 模块，名为 js-plugin，一共就 100 多行的代码，你可以直接使用；或者作为参考，自己动手实现这样一个引擎。项目的 github 地址是：https://github.com/rekit/js-plugin。

小结

在这节课，我们分析了软件复杂度产生的根源，来自复杂的依赖关系。随着功能的增加，系统复杂度也在不断增加，那么整个项目就会到达一个不可维护的状态。

所以我们首先需要从项目结构层面，去对复杂度做物理上的隔离，确保业务模块相关的代码都能在独立的文件夹中。

其次，我们要妥善地处理业务模块之间的依赖关系。不仅需要在业务上区分硬依赖和软依赖。同时呢，在技术的实现层面也要能做到模块的松耦合。

当然，上面的所有介绍要落实到实际的项目，还有很多细节问题需要考虑，比如如何避免在单点模块定义所有的路由，如何避免一个导航菜单组件包含了所有业务功能的导航逻辑，等等。

总结来说，这节课我们主要介绍的是**整个隔离复杂度的思路**，你可以根据实际场景进行有针对性的思考，进而解决复杂度的问题。

同时更为重要的是，在进行实际项目开发，尤其是大型项目的架构设计时，一定要时刻有**管理系统复杂度的意识**，不能只考虑功能是否实现，而不管复杂度，那样终究会导致系统越来越复杂，不断降低开发和维护的效率，甚至导致项目失败。

思考题

如果项目使用了 Redux，那么一个应用一般全局只有唯一的 Store，然后通过 dispatch action 和 reducer 去管理这个 Store。那么采用了按领域组织的文件夹结构后，怎样才能让业务功能相关的 Redux 代码也实现在各自的文件夹下呢？

欢迎把你的想法和思考分享在留言区，我会和你交流。同时，我也会把其中一些不错的回答在留言区置顶，供大家学习讨论。

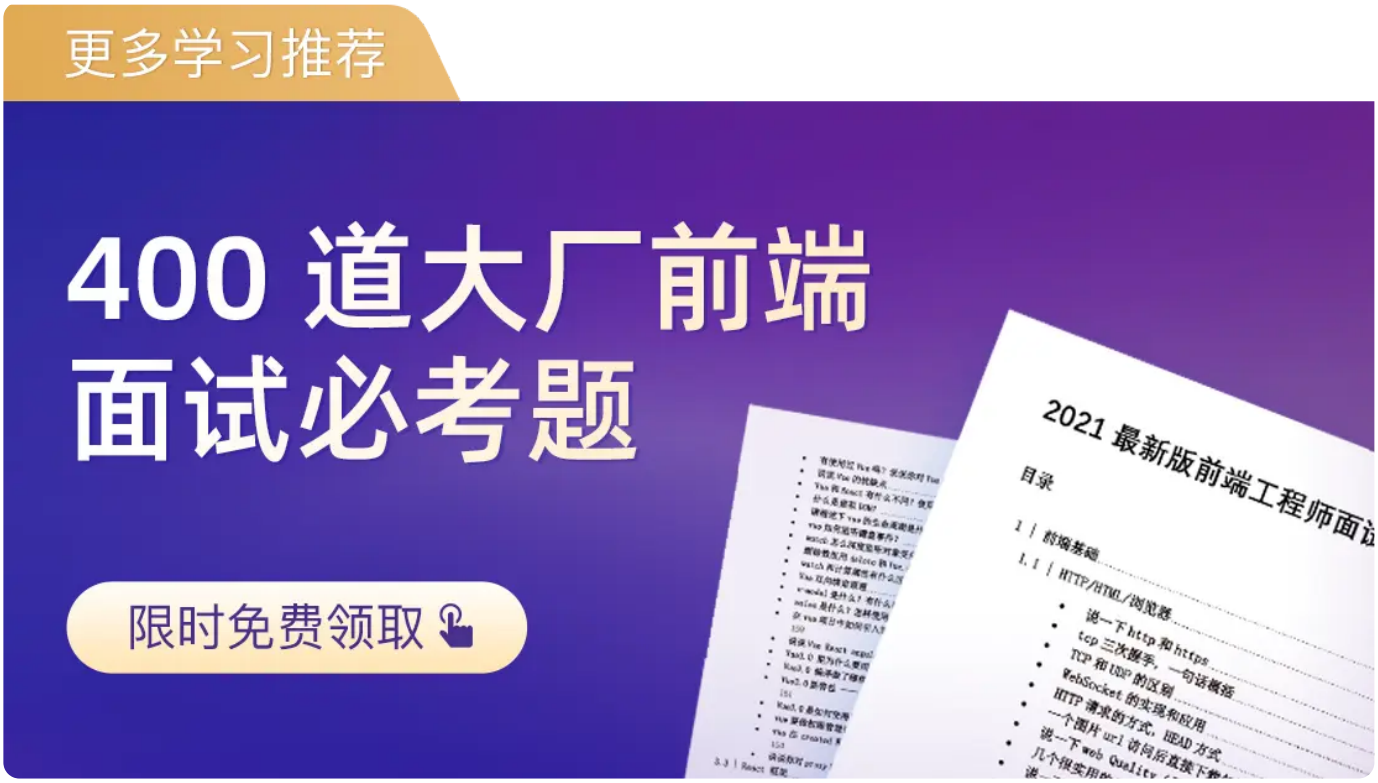
分享给需要的人，Ta订阅后你可得 20 元现金奖励

赞 4 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 11 | 事件处理：如何创建自定义事件？

下一篇 13 | Form：Hooks 给 Form 处理带来了哪些新变化？



精选留言 (13)

写留言

陈好人
2021-06-22

用一个注册的方式实现控制反转，确实是可以避免硬链接。不过有另一个问题，比方说我就不能直观看到article的组成了，同时，调试上会不会引来新的复杂度？

展开

作者回复: 关于 article 的组成，在我们项目中会有静态的分析工具，能够直观的看到一个扩展点被哪些模块进行了扩展。也能看到一个 feature 内部，使用了哪些扩展点。所以这个问题是可以解决的。

在调试上，因为扩展点本身和 article view 这个组件是几乎没有直接联系的，只是通过传递了一些参数。所以调试时只需要调试扩展模块就可以了，实际的使用不会有调试上的困难。

1 5

小个子外星人：）
2021-06-26

没有太理解，下次再看一遍

展开

3

何用
2021-06-23

Extension Point 可以用广告位招租形象地解释

展开

2

程
2021-06-22

js-plugin的用处感觉比较大。
但是在react组件中，<Extension name="article.footer" args={article} /> 这样的写法，确实我不太能理解。就像是一个容器组件，通过传入不同的参数，返回不同的组件。因为我使用ts，感觉这样又少了类型检擦。

2

SenjougaharaSama
2021-07-14

Extension 有用但用处很小，解决场景是删除被依赖模块时直接删除原模块文件，而不需要改动依赖文件。但如果依赖文件就只依赖这一个模块，那删除模块后，一样需要但依赖文件中移除extension ，理论不用删也能运行，但这多余的代码以及对其他人review 都增加了困惑

作者回复: 并不是多余的代码，它让已有模块具备了扩展能力。不意味着一定有人去扩展。

1

焱焱
2021-07-07

这章没怎么理解 希望老师能加餐再讲一讲 另外如果配合ts应该怎么扩展这个功能呢

1

何用
2021-06-23

extension point 的思路可以，将事件通信的机制用在了组件渲染的场景。register 就好比注册事件，Extension 就好比派发事件，依次将register的组件渲染在指定区域，但也不无缺点，比如对类型支持不友好

1

发芽的紫菜
2021-06-30

我们管理中心每一个页面都是一个目录，里面有自己的store ，api，components等，公共模块的话就抽出来，页面间没什么关联，改起来确实放心，不过冗余也多

展开

Aaron
2021-06-29

这章讲的很棒，感觉和依赖注入的思想有些相通之处，避免直接处理依赖，Angular 适合开发复杂应用也是因为这个特性更容易控制项目复杂度

展开

桃翁
2021-06-29

对于各个feature 可能会用到的 component和hooks 还是放到跟feature同层吗？

王昂
2021-06-26

老师这章不太理解，网上也没找到太多的例子。

展开

琪琪
2021-06-24

这章没看懂呢 老师

展开

喵咪爱吃肉

2021-06-24

老师，这个演示结构我还是不能太理解，能否给一个具体的项目做示例呢？